UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) University of Washington Department of Psychology Seattle, Washington 98105	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

A METHOD FOR BUILDING DATA MANAGEMENT PROGRAMS

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Scientific Interim

5. AUTHOR(S) (First name, middle initial, last name)

Earl Hunt and Gary Kildall

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
December 30, 1970	20	8

8a. CONTRACT OR GRANT NO. AFOSR-70-1944	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO. 9778	
c. 61102F	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d. 681313	AFOSR - TR - 71 - 2853

10. DISTRIBUTION STATEMENT

Approved for public release;
distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
TECH, OTHER	AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (NL) 1400 WILSON BLVD ARLINGTON, VIRGINIA 22209

13. ABSTRACT

Data management is usually done through a set of subroutines, called a kernel package. The programmer using or modifying a system designed with the kernel package need only grasp the few simple concepts and operations involved in the kernel. This approach has been applied in the construction of three substantial applications; a conversational version of computer language, a generalized information retrieval system, and a system for graphics based information retrieval

DD FORM 1 NOV 65 1473

UNCLASSIFIED

Security Classification

A [redacted] METHOD FOR BUILDING DATA MANAGEMENT PROGRAMS

Earl Hunt
University of Washington

and

Gary Kildall
U. S. Navy Postgraduate School
Monterey, California

This research was partially supported by the National Science Foundation, Grant NSF B7-1438R, the United States Air Force Office of Scientific Research, Air Systems Command Grant AFOSR 70-1944 and by the University of Washington Institutional Research Fund.

Computer Science Group
University of Washington - Seattle
Technical Report No. 70-12-09
December 30, 1970

A [redacted] METHOD FOR BUILDING DATA MANAGEMENT PROGRAMS[1]

Programs to manage large amounts of intricately structured data are hard to write. They are delivered late and usually contain many bugs when delivered. Only indifferent success has been obtained from "generalized" data management systems. The usual way to write large data management programs is to obtain the services of several wise old (or bright, young) hands and hope that they will read Knuth (1969) and Lefkovitz (1969) before setting out to reinvent the wheel. Usually this hope is not fulfilled. The resulting code is often so replete with programmer specific or problem specific tricks that it is hard to document, maintain, or modify.

We shall describe a method for the rapid construction of data management programs. The basic idea is that all data management is done through a set of subroutines, called a <u>kernel package</u> which we have found helpful. The programmer using or modifying a system designed with the kernel package need only grasp the few simple concepts and operations involved in the kernel. The approach is somewhat akin to the approach used to build complex gadgetry out of Heathkit or Digibit components, hence the name of the paper. Our approach has been applied in the construction of three substantial applications; a conversational version of the APL language, a generalized information retrieval system, and a system for graphics based information retrieval. At least one other organization has also used our approach successfully. We shall first describe the technique and then discuss the way it was used in each application.

<u>Basic</u> <u>concepts</u>. The term <u>Level</u> <u>0</u> storage will refer to word addressable, fast access store (usually core memory), and <u>Level</u> <u>1</u> storage will refer to file oriented random access store, normally a drum or disk. We assume a system which has both level 0 and level 1 store. The ideas could easily be generalized to a multi-level system. We also assume a user who writes a program to manipulate data in level 0 store, using the FORTRAN or ALGOL languages, but who has a very large amount of data that is to be held in level 1 store and requested on demand. The kernel package is used to relieve the user from worrying about either the management of level 1 store or the interplay between level 0 and level 1 store. The kernel also provides the user with commands for creating and manipulating very complicated data structures without ever being forced outside of the confines of FORTRAN or ALGOL.

<u>Data</u> <u>structures</u>. The user's data resides in a named file in level 1 storage. The kernel system divides the file in two ways. Physically the file is organized into <u>pages</u>, which are brought into level 0 as needed. The kernel system keeps track of the location of pages in both level 0 and level 1 store. Logically, the user's data is organized into <u>units</u> which he manipulates directly. Insofar as the user is concerned, a unit is a logically connected set of records in one of two types of formats. If the unit is an <u>ordered</u> <u>storage</u> <u>unit</u> its entries are organized by fixed formats called <u>fields</u>. These are established when the unit is created. One of the fields is designated as the <u>sort</u> <u>field</u>. Within an ordered storage unit records are kept sorted on the sort field. <u>Sequential</u> <u>storage</u> <u>units</u> contain variable length character strings, placed in the unit wherever room for them can be found at the time they are inserted. Ordered storage units are

typically used to provide indices by which particular strings may be found in sequential storage units. A simple example is shown in figure 1.

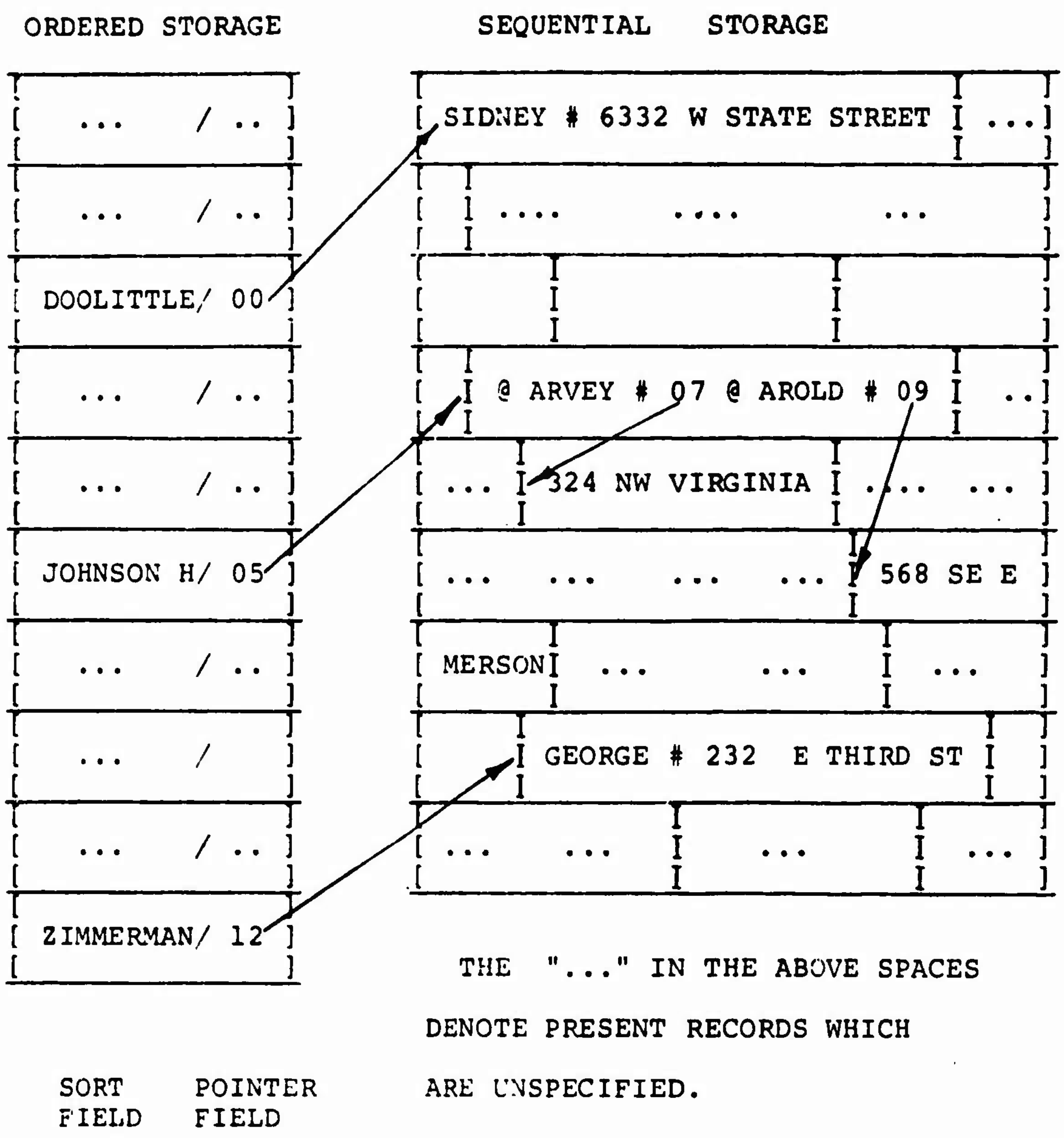
Units are identified to the user's program by unit number. A unit is created (and space reserved for it) only when the user's program calls the appropriate kernel subroutine. Pages are created for units on demand. Page creation and manipulation is done entirely within the kernel system, so the user has no way of controlling or interfering with it.

System use. The subroutines within the kernel system are listed in Table 1. They fall into three major groups. Storage maintainence subroutines communicate between the user's program and the computer's operating system, and declare and release units. They also allow the user a small amount of control in balancing the use of physical resources. The user may specify the amount of level 0 memory to be reserved for pages (thus decreasing the number of references to level 1 storage to get data needed by the user program) and he may specify times at which page and unit assignments in level 1 store are to be re-examined to see if better storage utilization can be obtained by shuffling data between pages. The user does not control the process of examination.

Storage interrogation and alteration commands allow the user to add, retrieve, or delete items to, from, or in units. The items are defined by naming character strings, simple variables, or arrays in the user's program. Finally, utility routines are provided to let the user determine the status of the data created by the kernel system.

The paging method. Let us shift from the user's view of the system as a package of subroutines to the systems program view, as a demand paging system for data.

FIGURE 1



THE "..." IN THE ABOVE SPACES DENOTE PRESENT RECORDS WHICH ARE UNSPECIFIED.

ORDERED STORAGE PROVIDES ENTRY INTO SEQUENTIAL STORAGE

TABLE 1

Storage maintainence commands

1. NAME identifies the user's file to the kernel routines.

2. SEQUENTIAL designates a declared unit to be a sequential storage unit. Unspecified units are assumed to be ordered storage.

3. ALLOCATE indicates the amount of level 0 storage to be used by the paging system.

4. MAINTAINENCE orders that the assignment of level 1 space be examined to determine if more efficient balance can be achieved between pages.

5. WRAPUP orders that file maintainence be completed. It is used at system termination of a user's run.

Storage interrogation and alteration commands.

1. STORESEQ stores a character string from the user's program into a specified sequential storage unit.

2. SEARCHSTORE places a character string provided by the programmer in a specified ordered storage unit in sorted order.

3. SEARCHORD searches an ordered storage unit for a specified record. The location of the closest match found is returned.

4. STOREORD stores a string of characters in an ordered storage unit in a specific location.

5. CONTENTS retrieves a record from a specific, programmer provided location in a sequential or ordered unit.

6. DELETE removes records from ordered or sequential units.

7. RELEASEUNIT orders the kernel program to release to the garbage area all pages attached to a particular unit.

Storage utility commands.

1. NEXTUNIT returns to the user program the unit number of the lowest unassigned storage unit.

2. SIZE returns the number of records in a specified unit.

3. UNITMODE returns the mode (sequential or ordered) of a specific command.

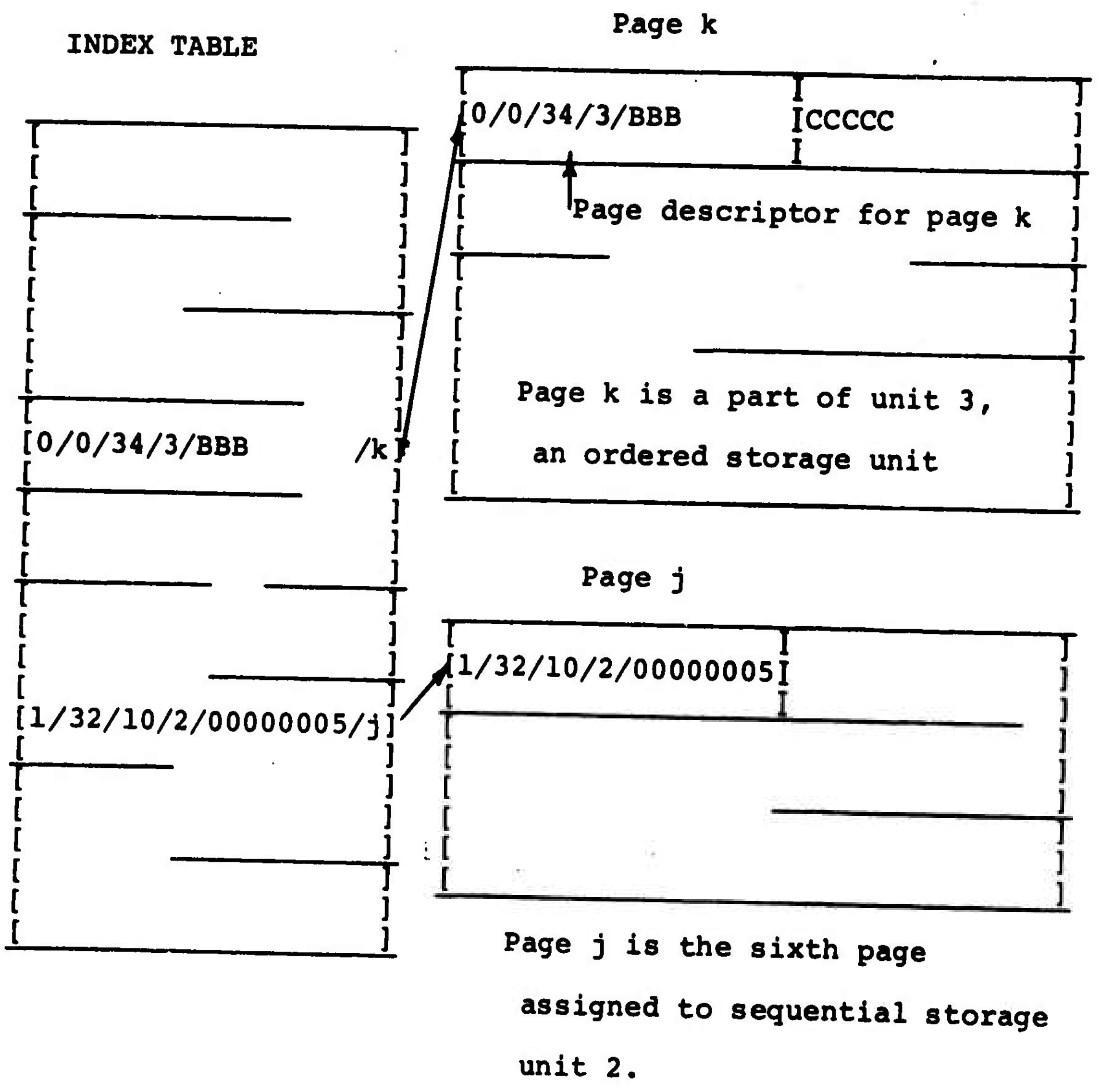
When the user first initiates the system he names a Level 1 file. The kernel divides the file into pages. Each page consists of two parts; the page descriptor and the page contents (user information). The page descriptor indicates the physical layout of user information in the contents area, including the size of the free space area at the end of the page. The unit number and type of the unit to which the page is assigned is also in the page descriptor. Any number of pages may be assigned to a unit. Assignment is done by the kernel system in response to user needs.

The kernel system keeps track of page assignments through the use of an index table in level 0 store. The entries in this table for all pages assigned to a particular unit are stored sequentially. The table entries are descriptors of the page. The relation between the index table and the pages is shown in Figure 2.

In addition to the index table the kernel system maintains a Unit Table which specifies the first and last entries in the index table assigned to each unit. The Unit Table is also kept in level 0 store.

A basic principle of kernel system operation is that pages should be shuffled as little as possible. Therefore, when a page is assigned to a unit a certain amount of blank free space is left at the end of the page. As more data is moved into the page, the free space is used as a buffer to avoid page overflows. If page overflows do occur new pages are allocated. One of the major jobs of the MAINTAINENCE subroutine is to examine pages, create new pages, and shuffle data between pages to avoid an imbalance between contents and free space that may have arisen as data was manipulated. This is done without moving data from level 1, by using

FIGURE 2


INDEX TABLE
Page k
0/0/34/3/BBB
CCCCC
Page descriptor for page k
Page k is a part of unit 3, an ordered storage unit
0/0/34/3/BBB /k
Page j
1/32/10/2/00000005
1/32/10/2/00000005/j
Page j is the sixth page assigned to sequential storage unit 2.

RELATIONSHIP BETWEEN PAGE DESCRIPTORS AND INDEX TABLE descriptors in the index table. Having a free space area minimizes the number of times maintainence needs to be requested by the kernel system itself. Ideally maintainence is limited to the post run WRAPUP or to times indicated by the user's program. This is particularly useful in conversational applications, since MAINTAINENCE can be called while the program is awaiting input from a console. There is no chance of the system becoming unresponsive, for the MAINTAINENCE subroutine (unlike WRAPUP) returns periodically to see if the program which called it wishes to go on to another task.

At run time copies of one or more pages may be held in level 0 store. The exact number of copies held is determined by the user, through the ALLOCATE command. The kernel system keeps track of the names of pages in level 0, so that there are no unnecessary accesses to level 1. In general it pays to have at least two pages in core, so that one may have available portions of a sequential unit and portions of an ordered unit serving as an index to the sequential unit.

This concludes our brief description of the kernel program. Kildall (1969) describes it in much greater detail. The original kernel was written in Burroughs Extended Algol for the Burroughs B5500 and is inextricably tied to that machine. Two of the applications we will describe are B5500 programs. A second kernel has been written in FORTRAN IV. We have attempted to make it "machine independent" except where it interfaces with the operatings system. For instance, the FORTRAN kernel must have the machine word and character size specified to it. The FORTRAN kernel has been tested and used for an application on the XEROX Data Systems Sigma 5, a 32 bit, byte oriented machine.[2] Subsequently we plan to test it on other machines.

Applications

B5500 APL. A conversational APL interpreter has been written for the Burroughs B5500. The system is similar to APL/360 (Falkoff and Iverson, 1968) in its outward characteristics. It is a multi-user, conversational computing system operating as a user program under the Burroughs B5500 Master Control Program. B5500 APL is internally divided into several components; a resource management section which schedules work for the other components, a terminal message handler for input and output, a monitor command and function editor section through which the user defines, edits, and traces the execution of APL functions, and a compiler-simulator section which translates from APL code to the order code of an hypothetical APL computer and then simulates the action of that computer. These components are apparent to the user. Quite hidden from him, but of central concern to use, is the virtual memory management section, which controls the allocation of user space in level 1 store. This section is not specialized to APL. It could equally well be used for any conversational computing application where the user needed the illusion of having a very large memory.

When an APL user enters the system for the first time he is assigned two storage units; an ordered unit called his name table and a sequential unit called his data table. Since the system is designed for multiple users, the numbers of these units cannot be predicted in advance. From the viewpoint of the kernel system, then, the APL system program is the user and the APL user simply an input source for the user program. The APL system calls for units from the kernel as it needs them.

In addition to calling for units when a user enters the system, APL must call for units through its function editor, as the user builds his library of programs and data. The APL user can declare three types of names; scalar names, array names (for numeric or character arrays), and function names. All names, regardless of types, are given entries in the name table. If the name is the name of a scalar variable its current value is also kept in the name table. If the name is an array name, the name table entry contains a pointer from the name table into the data table, where the string of characters defining the array value is located. When the name is the name of an APL function, the situation is more complicated. Upon user declaration of a function two units are created for it, the _function label table_ and the _function test table_. The label table is an ordered unit containing two types of entries. There is one entry for each line of text in the definition of the APL function definition. The line entry contains pointers to the line definition, which is stored as a character string in the function text table.

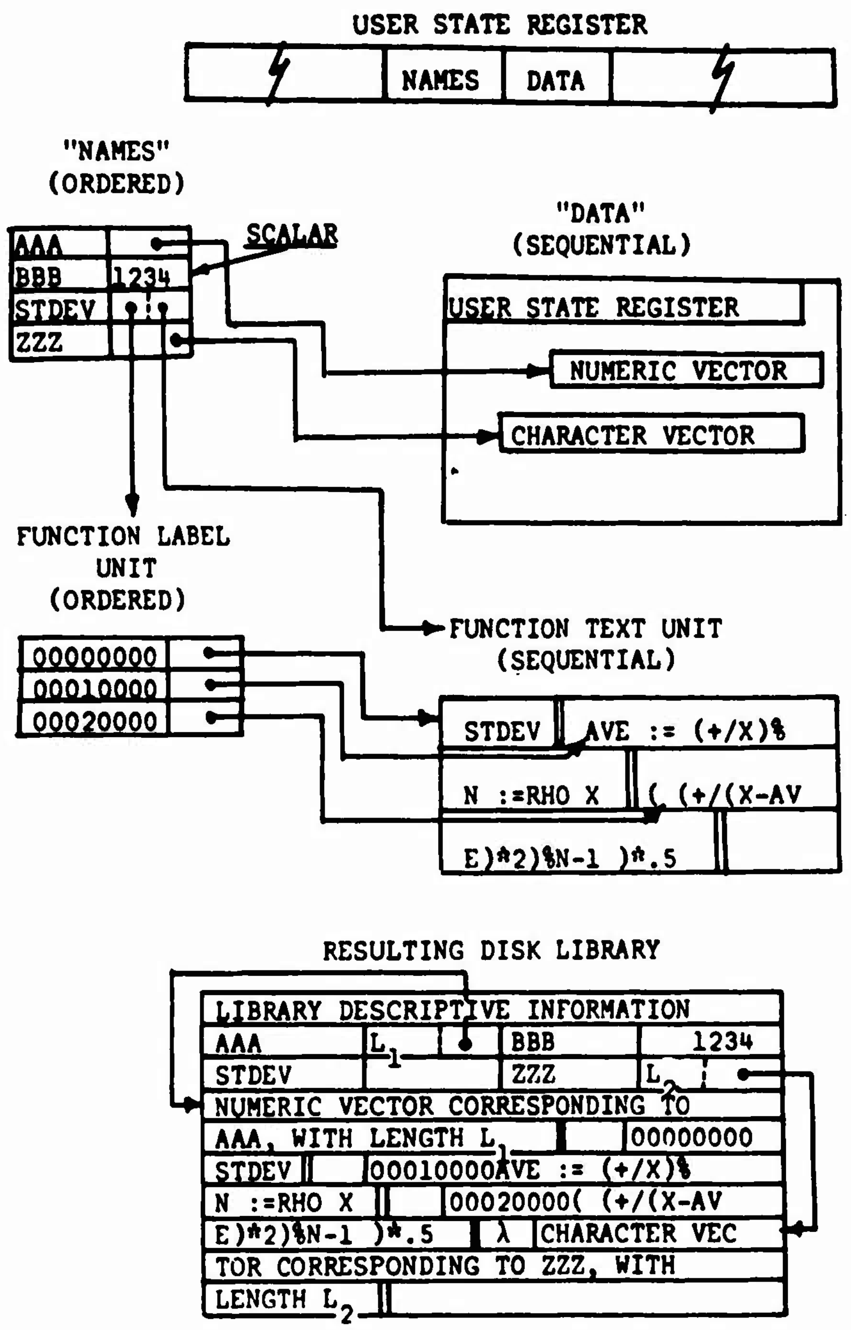
Recall that from the viewpoint of the kernel, the APL system itself is the user, and hence owns the file which the kernel is asked to organize. The result of the above process is that the kernel creates on the APL level 1 file a virtual memory for the APL user, containing his APL program and the value of all his variables. Because the size of the units assigned to a user can be expanded as needed (so long as there is room on the APL file itself) by causing the kernel to create new pages, the APL user has the illusion of having a very large machine. The way this machine is laid out in units is shown in Figure 3. The unit arrangement is transparent to the APL user, just as the page arrangement is transparent to the APL system programmer.

To operate the APL system must have an entry into the units assigned to each user. This is provided by a user state register (shown in Figure 3) which, among other things, contains the unit numbers of the user's name and data table. This is all the system needs to retrieve any piece of information about the status of the user's virtual memory. In practice, the user's APL program is executed by bringing appropriate bits and pieces of it from the level 1 file into a level 0 scratchpad area, where data is presented to the APL compiler and simulated APL computer. Note that because of the elaborate entry system into user virtual memory provided by the kernel program, only a very small amount of APL program need be brought into level 0 at any one time. For example, functions may be retrieved one line at a time. This greatly reduces the demand on costly level 0 memory without restricting the size of the program a user may write in APL.

A second interesting result of using the kernel system to keep virtual memory in level 1 store is that the system is very hard to disrupt due to computer system crashes. The APL system keeps in its level 1 file the equivalent of a program counter for virtual memory. This program counter is updated whenever new data is moved to or from the scratchpad to level 1. When the machine crashes the level 1 APL file will contain sufficient information to restart the user from the point of the last level 1 access before the crash. Typically, then, the amount of computing the user will lose is measured in milliseconds. This is no small advantage to any interactive computing system.

9a

FIGURE 3



THE FORMAT OF A LIBRARY

The B5500 APL system, and with it the Extended Algol kernel program, have now been in operation for over a year. We regard this as a stable application of the Heathkit technique.

A generalized Information Retrieval System. The B5500 Extended Algol kernel has also been used to construct a program called IRSYS, for defining and operating information retrieval systems (Finke, 1970 a,b). Like the APL system, IRSYS is best thought of as a program by which the ultimate user defines his application, and not as an application program in itself. Again like the APL application, the IRSYS programmer is quite unaware of the kernel system, although the IRSYS program writer uses kernel continually.

Externally IRSYS looks like any number of other information retrieval systems. Its basic user unit is the data set. A data set consists of a reserved data set symbol, followed by one or more data set elements, and an end symbol. A data set element consists of an element symbol and an element value. The element value may be either a number or a character string. For example, a data set defining a book might be written

```
/DSET
$AUTHOR MEADOW C.T.
$DATE 1970
$PUBLISHER WILEY
$COMMENT TOPICAL TITLE
/END
```

The terms /DSET and /END are the data set symbol and end symbol, respectively. $AUTHOR, $DATE, etc. are element symbols, and the strings following them element values. A data set may have more than one element of the same type.

IRSYS is used to define a file consisting of such items and to retrieve items referenced by the values of different elements.

A user interacts with IRSYS in three ways. In <u>definition</u> <u>mode</u> the user states what the reserved symbols will be. Elements are defined to be retrievable with character string values, as $AUTHOR in the example above, numerically retrievable, with numbers as values as $DATE in the example, or miscellaneous, non-retrievable elements ($COMMENT above). IRSYS accepts these definitions and reserves the necessary tables for the user dictionary by calls to the kernel system. Unlike APL, IRSYS is a single user program, so the kernel is used to organize a separate IRSYS file for each user.

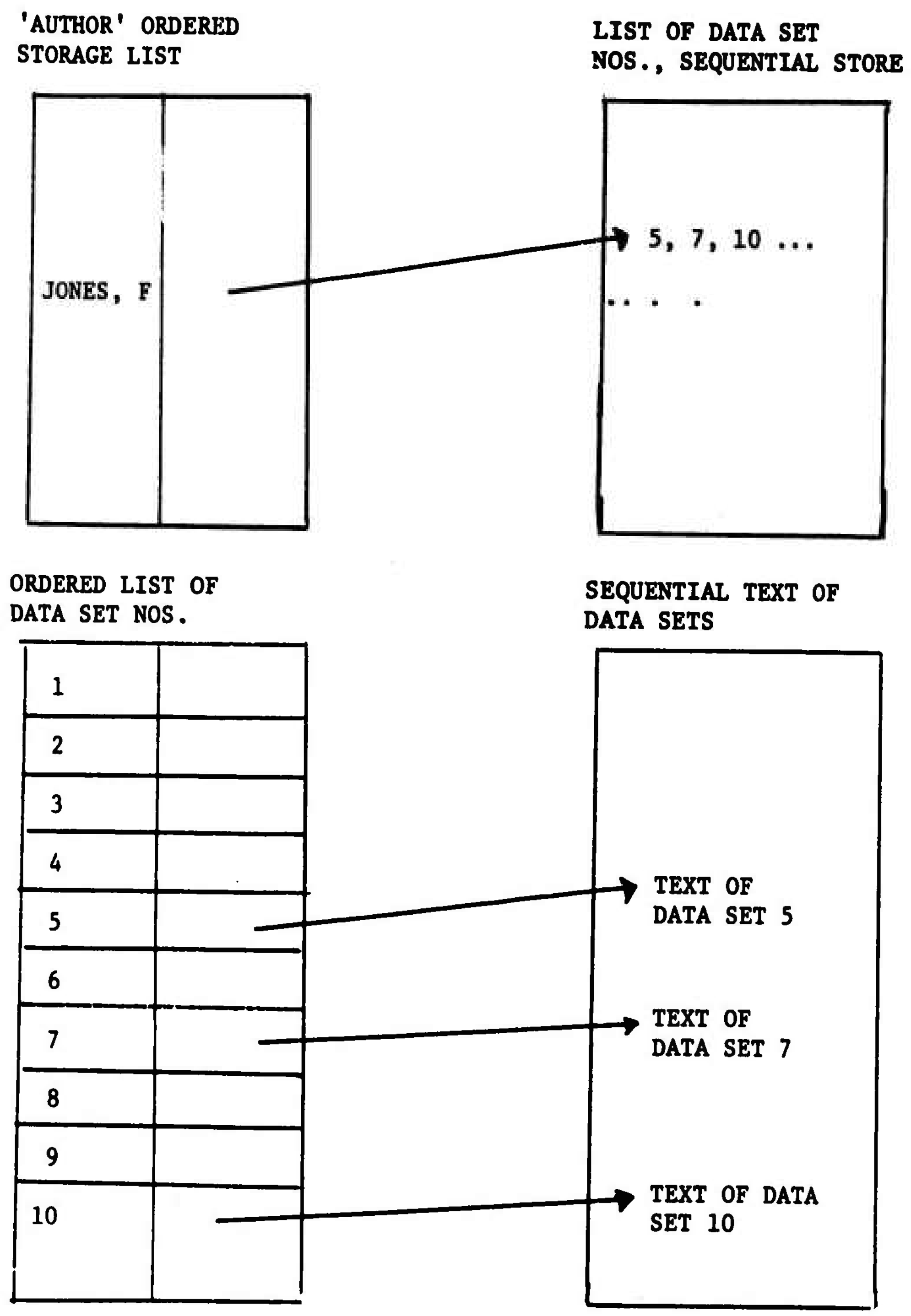
In <u>retrieval</u> <u>mode</u> the user writes queries above data sets by expressing Boolean combinations of statements about the values of retrievable elements. Relational statements may be used to reference values of numerically retrievable elements. Thus the query for (($PUBLISHER WILEY) AND ($DATE .GT. 1960)) refers to all data sets with WILEY in a PUBLISHER element and with a $DATE element greater than 1960. IRSYS will locate the data sets specified by a query and report their number. It will print these sets only on command, either on the line printer or on a console.

In <u>storage</u> <u>mode</u> the user stores data sets into his IRSYS file and edits data sets already in the file. Obviously one definition session must precede any other session, and at least one storage session must precede the first retrieval session. Otherwise there is no fixed order to the sequence of sessions in each mode. IRSYS can accept input either from a remote console, as an interactive IR system, or from the card reader in batch mode. A file established in batch mode may be interrogated from a console and vice versa.

Now let us look at how IRSYS uses the kernel system. Only a few examples of the technique will be given, since there are many special uses to allow for "pathological" user definitions such as one name's being contained in another. Finke (1970a) discusses all applications of the kernel with considerable detail and clarity.

Each retrievable element symbol has associated with it an ordered storage unit which is created when the retrievable element is defined. The value entry contains both the value name and a pointer to a second unit in which the strings are the internal numbers of data sets which have the appropriate element and value. Whenever a data set is entered in storage mode it is assigned an internal number. Its elements are then examined. The element name table of each element in the data set is examined to see if the element value has appeared before. If it has, there will be an appropriate entry in the element table, which will point to a sequential unit. This unit contains a list of previous data sets which have the same element and value as the new data set. The internal number of the current data set is added to this list. If the element value has never appeared before a new entry is made in the element name unit (in the proper place in the sequence) and a new, one entry list is created in the sequential storage unit. The result is a set of cross index tables pointing to lists of data set numbers, as shown in Figure 4. In addition, when the data set is entered its entire text is converted to an internal form (from which the original form is recoverable), and is placed in a sequential unit called the DATA SET Table. An entry for the

FIGURE 4



ORGANIZATION OF DATA SET FILES

IN IR SYSTEM

data set is also made, under its internal number, in the Data Set number table. The Data Set number table is an ordered storage unit whose entries point to the appropriate text in the Data Set Table. The result is shown in simplified form in Figure 4. Not shown is another ordered set in which IRSYS keeps a list of the element names and the number of their associated units.[3] IRSYS uses the kernel to keep this list and many others besides. By examining Figure 4 one can see that given an element value, IRSYS has the capability of finding data sets containing that element.

In interactive information retrieval applications the user must often sharpen his question before he locates the set of items he really wants. For example, suppose we were interrogating a file of cinema reviews. The question "$RATING INDECENT" might locate 4970 data sets, while the question $RATING INDECENT AND $DATE .GT. 1965 might find only 100[4]. It would be more efficient if the questions were asked in series, and the search for $DATE .GT. 1965 restricted to the set of data set numbers already retrieved in response to the first question. To allow for such situations IRSYS permits the user to "nest" his questions, first asking a question which refers to a set of data sets, and then asking questions which are understood to refer only to that set. The kernel system is used to maintain the various temporary lists needed by IRSYS in this exchange.

IRSYS has now been in operation about nine months without maintainence. It has produced useful results for a number of users.

Graphic Information Retrieval. Our final example is different in a number of ways. The application involves the use of an XDS Sigma 5 to control a graphic display device known as an ARDS[5].

A MAP MANIPULATOR program has been written in Fortran, using a Fortran IV kernel, to allow the user to draw a map (or portions of a map) on the face of the ARDS display. The map is a conventional street map, composed of streets, barriers, areas, and points of various types. Messages may be associated with any map element. Speed indicators are associated with area and streets. The MAP MANIPULATOR program is not, itself, intended to do anything useful. It is intended to be a component in a larger system for displaying information in a "command and control" situation in which the user must observe and direct moving units, such as police patrol assignments or air traffic control.

Initially the user "draws" the map on the ARDS display face using the mouse. At any time he may edit the map or associate a message with an element of the map. The user indicates an element of interest either by "pointing" to it with the mouse or by referring to its internal number. (If the user does not know the internal number of an element he may obtain it by pointing and asking.) Once the user has established a map he may ask that different portions of it be displayed, at different magnifications, by zooming or windowing. Thus he has at his command roughly the capabilities of a simplified SKETCHPAD system (Sutherland, 1963), specialized for map manipulation.

The basic information unit of MAP MANIPULATOR is the MAP TABLE. This is a sequential unit whose records are strings in a language for defining map elements. Each sentence in this language must conform to a BNF syntactical specification. For example, the grammatical definition of a road is

```
<road>::=      1 <roadname>  <route>   |   <road> <route>
<roadname>::=  {positive integer}
<route>::=     <speed>   <line>
<speed>::=     {negative number}
<line>::=      {co-ordinate pair}   {co-ordinate pair}
```

The associated semantics are:

1 is an identification symbol to aid in interpretation. Remember that this "language" is for strings that will be read by a program, not a person.

The <roadname> integer is a pointer to an entry in the ROAD TABLE, an ordered unit that will be described below.

The absolute value of <speed> indicates a multiple of the basic speed. This is interpreted as the value of speed of movement along the road. It is intended for use in answering questions about best routes from one point to another, or about the transit time along a particular route.

The co-ordinate pairs for the <line> are the endpoints of a vector. In the MAP TABLE they are in map co-ordinates, i.e. they refer to the scale of the map, which may be much more than can be displayed.

ROAD TABLE is an ordered unit which contains one entry for each road in the map. This entry contains pointers back to the string defining the road in the MAP TABLE, and, if appropriate, to the string defining the road as displayed (in display face co-ordinates) on the sequential unit DISPLAY TABLE. In addition, a ROAD TABLE entry contains a pointer to a MESSAGE LIST table entry. MESSAGE LIST is a sequential unit whose records are lists of message numbers associated with a particular map element. Each message number names an entry in the ordered MESSAGE TABLE, which in turn contains a pointer to the message text in the sequential MESSAGE TEXT table.

The overall structure of the data defining roads in MAP MANIPULATOR is shown in Figure 5. Similar figures could be constructed showing tables for barriers, areas, pointers and events.

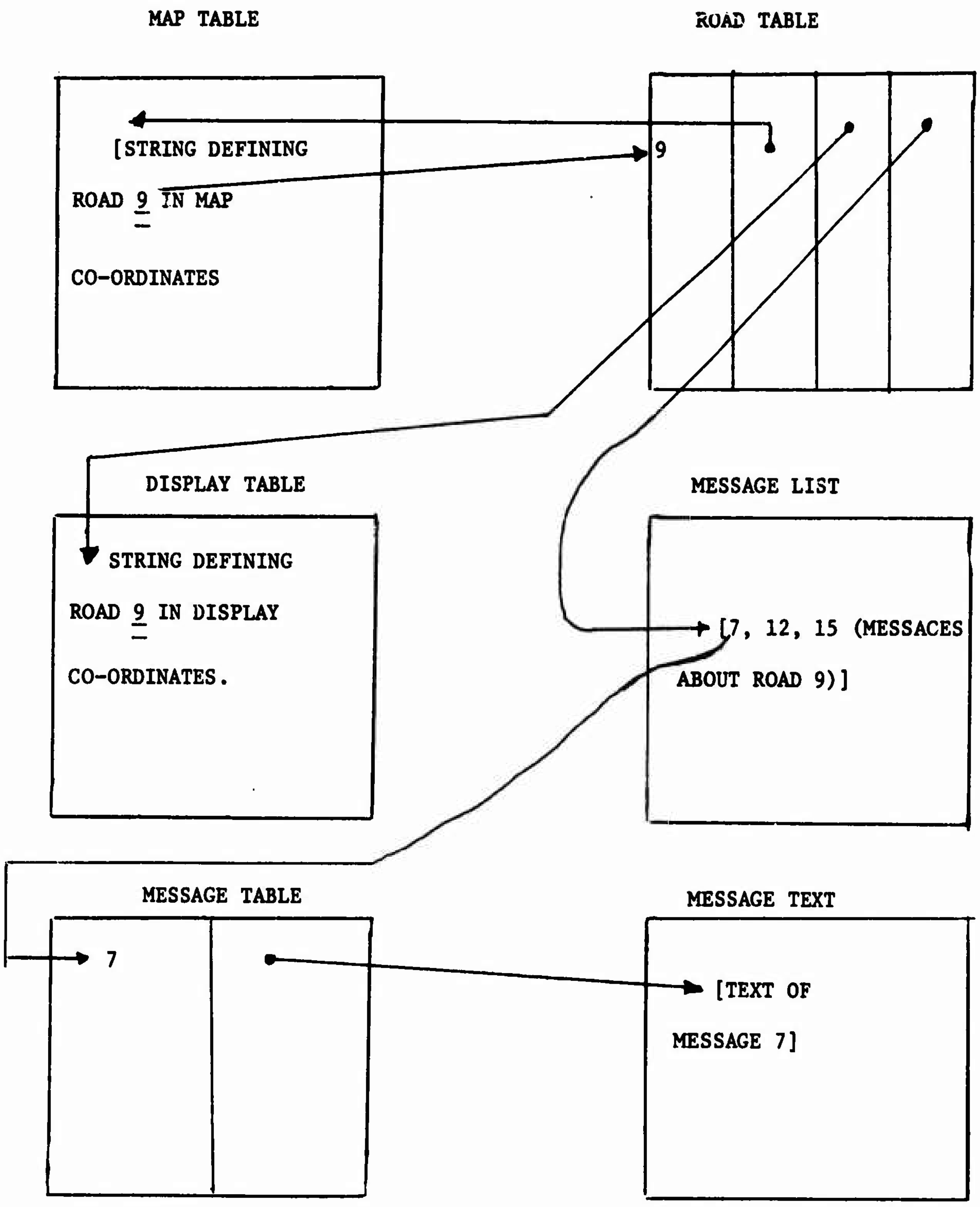
We have pointed out that MAP TABLE defines elements in terms of map co-ordinates. A DISPLAY TABLE unit, similar in form to MAP TABLE, defines the elements actually being displayed at any one time in terms of display coordinates. This is necessary both for windowing and zooming and to determine what element the user is pointing at with the mouse.

The structure of data in MAP MANIPULATOR gives us the potential for asking a number of questions about graphically displayed data. For example, one can display a map, indicate an area by pointing at it, ask that the area be displayed at an appropriate magnification, and either insert or retrieve messages about the area. Note that these messages could be entries into an information retrieval system similar to one established by IRSYS. We have not actually made this connection as yet.

Summary

Nothing we have reported is terribly exciting or new. Building complex systems requires sophisticated tools. The Heathkit approach is an attempt to make it easier to program complex information management systems, and to produce programs that are easy to maintain and understand. Basically, the approach will work if many systems can be designed with the same tools. It will not work if every information management problem is unique. We

FIGURE 5



STRUCTURE OF MAP MANIPULATOR

think the Heathkit approach works well. While we certainly do not want to disparage any of our colleagues who have worked on the various applications we have described, we think it is correct to say that none of them came to these projects with any substantial background in system programming. The timetable for the results was as follows:

B5500 was completed by one person working half time for six months.

MAP MANIPULATOR was brought to a nearly workable stage by two people working quarter time for three months. A substantial part of this time was spent checking the Fortran IV kernel system. The kernel system is far from simple, and obviously must work if anything else is to work.

We believe this is a good record. It may be, of course, that the resulting programs execute very inefficiently. It is hard to determine whether inefficiencies should be blamed on the Heathkit approach or on the fault in a particular implementation. In fact we are not at all sure that inefficiency does follow from our approach. IRSYS has been used by a number of people who have not complained about the bills. IRSYS contains routines to monitor system operation, and this is currently being done. APL B5500 appears to be reasonably economic if the system on which it runs is not too heavily loaded. In any case, execution efficiency is not the only criterion for system programming. There is a good argument for bringing up a working system quickly and then replacing it with a highly efficient one at your leisure. How many microseconds are there in a month?

Acknowledgements

So many people have aided us at various stages that our acknowledgements must perforce read like movie credits. We list them alphabetically by project.

APL project. Hellmut Golde, LeRoy Smith, Sally Swedine, and Mary Zosel.

IRSYS. Jeanne Finke.

FORTRAN IV Kernel. Thomas Kuffel

MAP MANIPULATOR. Jerry Jensen and Duane Sands.

And last, but not least, let us mention the first user who was willing to rely on a Heathkit built system (IRSYS) to solve his problem, although he did not have the slightest interest in advances in computer science. Such a user always discovers the bugs in the perfectly checked out system. Our user, Professor David Bonsteel of the University of Washington School of Architecture, was no exception. May his buildings never crash.

Footnotes

1. Portions of this research were supported by the National Science Foundation, Grant NSF B7-1438R, the United States Air Force Office of Scientific Research, Air Systems Command Grant AFOSR 70-1944 and by the University of Washington Institutional Research Fund. During the initial period in which the research was conducted, Gary Kildall held a National Science Foundation predoctoral fellowship at the University of Washington.

2. We are aware of a second FORTRAN kernel used to implement a business information system on a Digital Equipment Corporation PDP-10. We have examined this kernel and, although it is written in Fortran, we feel that it uses so many (quite effective) machine dependent tricks that it is solely a PDP-10 program.

3. The observant reader may have noted that if this were all that IRSYS did it would be unable to handle situations in which one term included another - e.g. JOHN and JOHNSON, if both terms filled up one of the fixed fields of an ordered storage unit. Such problems are handled in IRSYS by a rather complex system of pointers from ordered to sequential units, the description of which would add little to the current discussion.

4. O tempora! o mores!

5. This is a Computer Displays Incorporated Advanced Remote Display System (ARDS). The ARDS is essentially a storage type CRT, a keyboard, and a graphic input device known as a "mouse". By moving the mouse on a table the user can

manipulate a point or vector on the display face, which can then be located by the computer. In addition to the (hopefully) machine independent kernel system this application uses the systems software for controlling the mouse, ARDS, and communication equipment which is part of the operating system of the University of Washington Computer Science Teaching Laboratory (Hunt, 1970).

References

Finke, Jeanne M. A Generalized File Management System, U. W. Comp. Sci. M.S. thesis, 1970(a).

Finke, Jeanne M. A User's guide to the operation of an information storage and retrieval system on the B5500 computer. U. W. Comp. Sci. Tech. Report 70-1-3, 1970(b).

Hunt, E. The Computer Science Teaching Laboratory at the University of Washington. ACM, SIGCSE Bull. 2, 1970, 30-33

Iverson, K. and Falkoff, A. D. APL/360: User's Manual. IBM Corp., 1968

Kildall, G. A. Experiments in large scale computer direct access storage manipulation, U. W. Comp. Sci. Tech. Report 69-01, 1969.

Kildall, G. A. APL/B5500: The language and its implementation. U. W. Comp. Sci. Tech. Report 70-09-04, 1970.

Knuth, R. D. E. The Art of Computer Programming. Vol. 1 Menlo Pk, Calif. Addison-Wesley, 1969.

Lefkovitz, D. File Structures for on-line Systems. New York Spartan, 1969.